Homework, RNN

Assignment 1, what is ‘gradient clipping’?
Gradient clipping is most common in recurrent neural networks (RNNs). It has to do with managing exploding gradients. Since this sounds kinda vague, lets make a small (non RNN) example to make it more clear.
Gradient descent can be used for more than just neural networks. As long as you have a clear loss function, and the loss landscape is not too turbulent, it can be used for solving many other problems. Here, for example, it is used to find the square root of a number (the number 16 in this case):

```
import torch

nmbr = 16 # the number you want to find the square root of.
start = torch.ones(1, requires_grad=True) # initial guess

for epoch in range(10):
    loss = (start**2 - nmbr)**2 # the loss function
    loss.backward()
    start.data -= start.grad * 0.01 # applying gradients with lr=0.01
    start.grad *= 0
    print('epoch %i:'%epoch, float(start))
```

```
epoch 0: 1.5999999046325684
epoch 1: 2.4601597785949707
epoch 2: 3.439068555831909
epoch 3: 4.013091564178467
epoch 4: 3.9962520599365234
epoch 5: 4.00104284286499
epoch 6: 3.9997074604034424
epoch 7: 4.000082015991211
epoch 8: 3.9999771118164062
epoch 9: 4.000006198883057
```

As you can see, it only needs a few epochs to come very close to the correct answer. If we print the gradients, it shows that the gradients start large, and get smaller as it approaches the correct solution, as you would expect:

```
epoch 0: -60.0
epoch 1: -86.01599884033203
epoch 2: -97.89087677001953
epoch 3: -57.402286529541016
epoch 4: 1.683960199356079
epoch 5: -0.4790581464767456
epoch 6: 0.13354921340942383
epoch 7: -0.03744233027100563
epoch 8: 0.010498262010514736
epoch 9: -0.002929670736193657
```

This method also works for many other numbers (the square root of 12.5 → 3.535533905029297). However, something strange happens when we try to find the square root of larger numbers, as can be seen in the image below when trying to find the square root of 312:

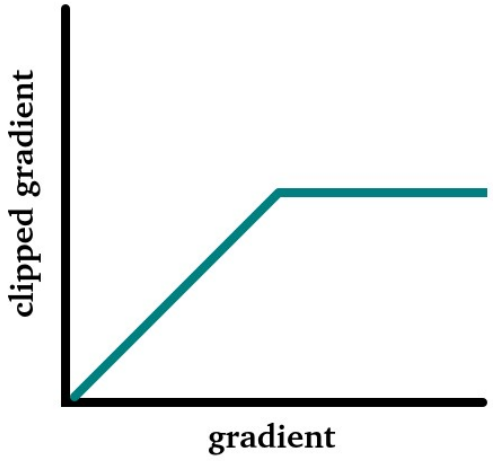
```
epoch 0: 13.4399995803833
epoch 1: 84.06257629394531
epoch 2: -22628.01953125
epoch 3: 463446179840.0
epoch 4: -3.981602641574079e+33
epoch 5: inf
epoch 6: nan
epoch 7: nan
epoch 8: nan
epoch 9: nan
```

corresponding gradients

```
epoch 0: -1244.0
epoch 1: -7062.2578125
epoch 2: 2271208.25
epoch 3: -46344622309376.0
epoch 4: 3.981602592056478e+35
epoch 5: -inf
epoch 6: nan
epoch 7: nan
epoch 8: nan
epoch 9: nan
```

Here you can see that the gradients are getting larger and larger while also swinging between positive and negative values. It is not hard to see why they call it ‘exploding’ gradients.

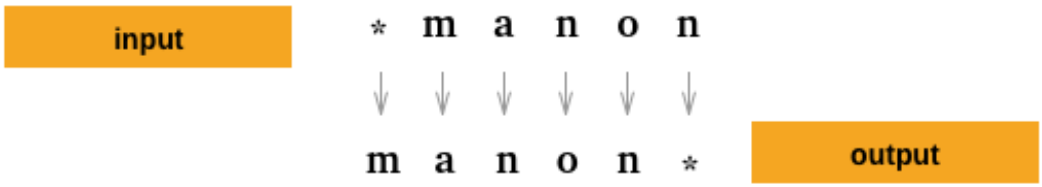
To prevent these kind of problems, the smart people behind RNN’s invented gradient clipping. As the name suggests, it cuts the gradients if it exceeds a certain predefined threshold:



And now the actual assignment:
Using the script above (in the folder under the name ***gradient_clip.py***), can you implement this 'clipping' and find an optimal threshold so that the script can find the correct solution of √312 (within an error of +- 0.25) after ~50 epochs?

Assignment 2, generative models, too few baby names to choose from
Some find it hard to think of new original names to give to their children. Lets see if a RNN can help creating some good new names! To accomplish this I have assembled a list of known country specific first-names (***finalNames.txt)***. Since there are clearly rules behind what defines a name (for example 'feffftttrheaaaa' is not a name in any language), we will try to let a RNN learn these latent rules. Once trained, it should be able to produce completely new original names given a specific country. With any luck, maybe some good ones :)

To make it work, we will feed the RNN one letter at a time. An internal memory will keep track of what is has seen before. The trick is to let the RNN always predict the next letter in the sequence as can be seen in the image below. Since neural networks can not process text directly, the letters will have to be converted to one-hot vectors. The output prediction will simply be a vector of probabilities for each possible letter (a vector with the same shape/size as the one-hot vector but with probabilities at each position). Since we are going to build a generative RNN, we add the '*****' symbol at the beginning as a 'starting symbol' so that it can also learn which letters are more likely to come first in a name, and a '*****' symbol at the end as 'stop symbol' so that we know the RNN is done generating the name.



I have created a RNN architecture which you can find on the last page. Since you've created neural networks before, I don't expect you will have any problem with creating one based on the image. Most of the RNN is pretty standard, just some linear layers, concatenation, addition, dropout and activation functions. The memory block is the only unique part. Here the memory will be a vector of zeros at the beginning of each word, and it will be modified after each letter (by adding one of the RNN outputs). You are free to experiment with the number of neurons per layer and for the memory. (As an indication, I used 128 neurons for all layers and the memory vector and relu as af).

Part 1, the annoying part:

next time will provide these helper functions so that students can focus on the RNN design part

Use the script ***rnn.py*** for the following tasks:

- Create a function letter2hot(letter) that convert a single string-character (letter) to a one-hot encoding. Shape → (1, nmb_letters)
- Create a function that convert a name into a tensor. Shape → (name_length, 1, nmb_letters)
- Create a function that convert a country-label into a one-hot vector. Shape → (1, nmb_countries)
- Import the names and their corresponding countries from ***finalNames.txt***
- Train the network → repeat until convergence: Take a random name from the dataset with the corresponding country. For each name initiate the starting memory as a vector of zeros (shape → (1, memory_length). Initiate the loss variable (loss=0). Feed the name letter by letter to the network. After each letter add the prediction error to the loss (loss += F.binary_cross_entropy(prediction, true_next_letter)). After going over all the letters in the name, calculate the gradients (loss.backward()), apply the gradients (optim.step()), and reset the gradients again (optim.zero_grad()). Don't worry about over-fitting the network, just let it train for a while.
- Create a function that takes as input a country name and outputs a RNN generated name. Do this by initializing the memory vector (shape → (1, memory_length) as a zero vector, and the first input letter as the one-hot representation of the '*' symbol. If fed with the country, the letter, and the memory, the network will output a probability distribution. Use the 'numpy.random.choice' function to sample from this distribution → numpy.random.choice(list(alphabet), 1, p = network_probability_output[0].detach().numpy()). Feed the network again with the one-hot version of the sampled letter (not the output probability!) and the modified memory vector to predict the next letter. Let the RNN keep predicting new letters until it predicts the 'stop symbol' ('*'). (don't forget to put the RNN in evaluation mode when generating and back to training mode once done)

Part 2, the fun part:

- Pick a country for name generation
- Generate ~30 new names. Do they look plausible given the country? (These should be original and never before seen first-names, judge them as such)
- Modify the name generator function (or create a second function if you wish). This function will make the most likely letters even more likely and the least likely even less likely. Do this by taking the square of the probabilities and dividing them by the sum of the squared probabilities before sampling a new letter.
- Generate again ~30 names using this new function. In your opinion, do they look better, worse or the same?
- In my personal opinion RNNs do a good job of generating names for a given country. But since some members from our group come from multicultural families (Jarek → Dutch+Polish, Daniel → Dutch+Chilean) I am curious if the RNN is capable of generating names that have a style resembling a combination of two countries. Lets do a experiment, pick two countries and generate again ~30 names using a representation of the two countries (example: [0, 0, 0.5, …..0, 0.5]). In your opinion, is the RNN able to generate plausible names that look a combination from the two selected countries?

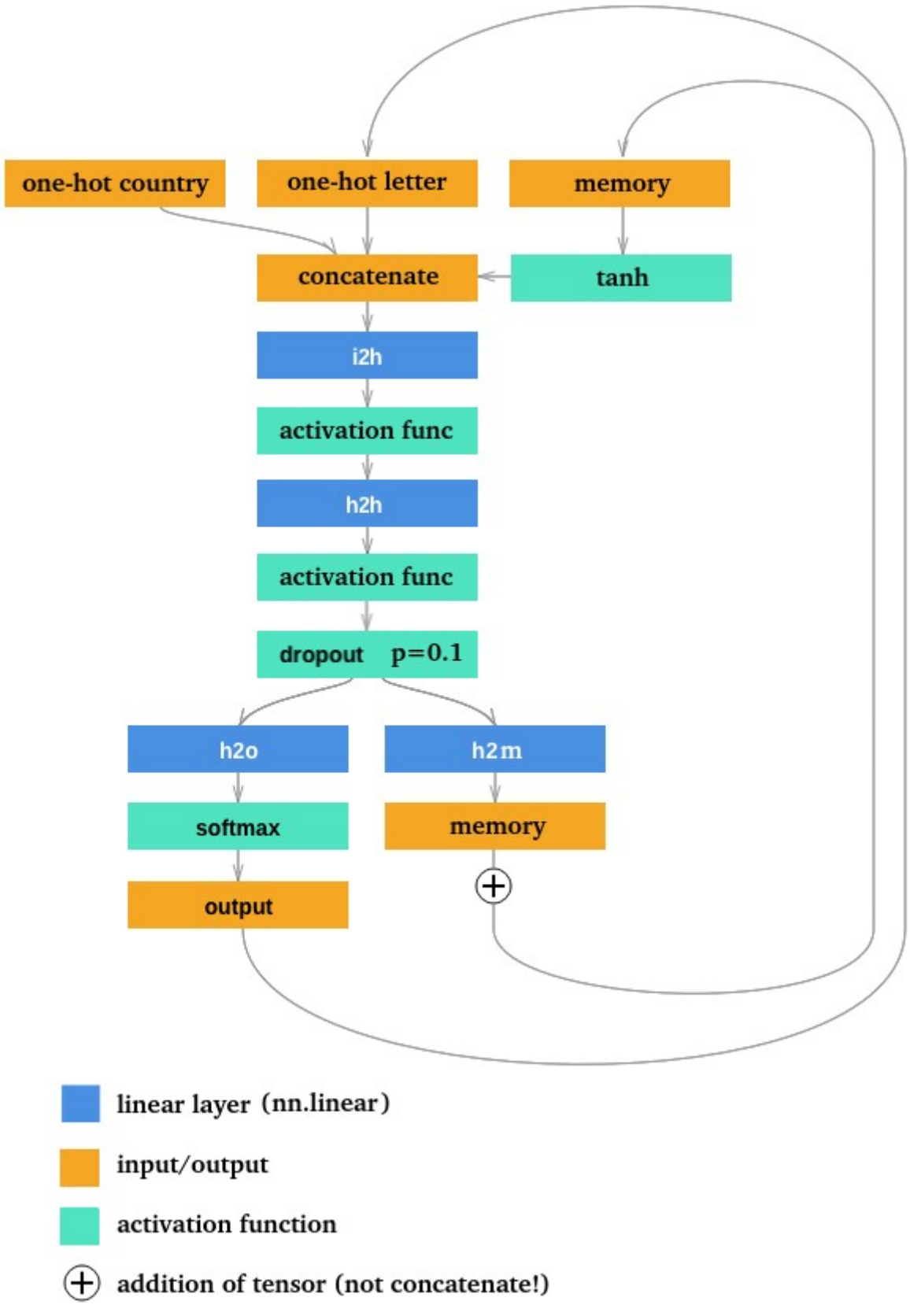
Instead of using loss: Each output is P(X_i | X_i-1). So multiplication of all output is the P(X1X2X3 | country)

- During training the loss was calculated based on a specific country and name. Therefore the network should learn which names are most likely for any given country. Once trained, a name that is unique for one specific country should produce a lower loss with the trained network than if the same name was given with a completely different country. So, the loss could indirectly be used as a classifier!

Given the countries **Netherlands, Iran, France, Poland, China and Italy** calculate for each of these countries the loss given your OWN name. Order the countries by loss, does the lowest loss indeed correspond with what you expect? (don't forget to put the RNN in evaluation mode!!!)

- Create a top 5 of favorite generated names (pick a favorite country, or combination) to share with the group.

Have fun, and good luck!


one-hot country
one-hot letter
memory
concatenate
tanh
i2h
activation func
h2h
activation func
dropout p=0.1
h2o
h2m
softmax
memory
output
linear layer (nn.linear)
input/output
activation function
addition of tensor (not concatenate!)